Lab 07: Python RegEx (Regular Expressions)

This lab gives practical examples to investigate Python for text manipulation. The main focus is on Regular Expression pattern matching; this is applied to extracting email addresses and IP numbers from email headers.

The lab also includes exercises for reading webpages with Python.

Regular expressions and reading webpages are essential components of the final coursework.

1. Pattern matching introduction

Regular Expressions (RegEx) provide Textual Pattern Matching. This is provided via the regular expression pattern matching language and the **re** module and its functions.

To search for a **single pattern** in a string the following syntax is used with the re.search() function:

```
Match_obj = re.search(pattern_str, str_to_be_searched)
```

2. Matching Text Characters

From the IDLE shell window, try the following.

Import the Standard library module re, and use the re.search() function, passing it a pattern and a string to search. It returns a match object:

```
>>> import re
>>>
>>> str1 = 'string to be searched for the word parrot'
>>> match = re.search( 'parrot', str1)
>>>
>>> match
>>> match.span()
```

Q: Has a match object been created?

Q: What information does the "span" in the match object hold?

Q: What type of object is match.span()?

Now try:

```
>>> if match:  print (f'found the pattern: {match.group()}')
```

Q: What does the match object's .group() method output?

If the pattern is matched, the match.group() method will hold the matched text from the string being searched.

Challenge: adapt the code examples above so that the print statement outputs:

```
Found: "parrot" between offsets 35 and 41
```

Q: What is the print statement used?

If the pattern is not matched, the match object will not be created. Try:

```
>>> match = re.search( 'seagull', str1)
>>>
>>> if match:  print (f'found the pattern: {match.group()})
```

Q: Has a match object been created?

3. RegEx: Matching Special Pattern Characters

The real power in using Regular Expressions is not to match on basic characters, but to define patterns to match. Typical examples of patterns are email addresses, IP addresses, postcodes, phone numbers, car number plates, etc. To allow us to match patterns like these, we use patterns representing ranges of characters – or 'character classes'.

Special pattern characters used include:

- **.** (dot) Matches any one character
- **\w** matches any word character (letter or number or underscore) - equivalent to [0-9a-zA-Z_]
- **\d** matches decimal numeric digits – equivalent to [0-9]
- **\s** whitespace – equivalent to [\t\n\r\f\v]

We can match letters or numbers with **\w**.

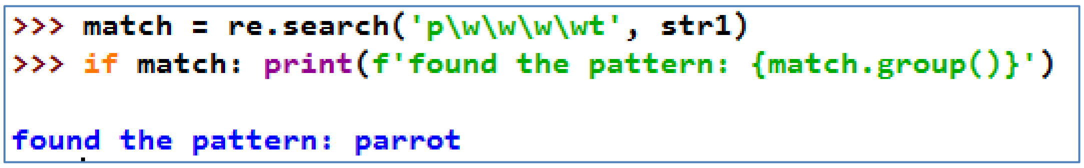
Try:

```
>>> str1 = 'string to be searched for the word parrot'
>>> match = re.search( 'p\w\w\wt', str1)
>>>
>>> if match:  print (f'found the pattern: {match.group()}')
```

Q: What has been matched against the pattern?

Q: Why?

Q: What is the correct pattern?



```
>>> match = re.search('p\w\w\w\wt', str1)
>>> if match: print(f'found the pattern: {match.group()}')

found the pattern: parrot
```

We can match numbers only with **\d**. Try:

```
>>> str2 = '1 string to be searched, for 23, the age of the parrot'
>>> match = re.search( '\d\d', str2)
>>>
>>> if match:  print (f'found the pattern: {match.group()}')
```

Q: What has been matched against the pattern?

Q: Why was the 1 not matched against?

Sometimes we need to match on <u>any</u> characters. Try:

```
>>> str3 = 'string to be s3@rch3d'
>>> match = re.search( 's\w\w\w\w\w\wd', str3)
>>>
>>> if match: print (f'found the pattern: {match.group()}')
```

Q: What has been matched against the pattern?

We can match any single character with the **. (dot)**. Try:

```
>>> match = re.search( 's......d', str3)
>>>
>>> if match: print (f'found the pattern: {match.group()}')
```

Q: What has been matched against the pattern?

We can match whitespace characters only with **\s**. Try:

```
>>> str4 = 'string to be 445 3665 searched for phone number'
>>> match = re.search( '\d\d\d\s\d\d\d\d', str4)
>>>
>>> if match: print (f'found the pattern: {match.group()}')
```

Q: What has been matched against the pattern?

Try replacing the space with a tab character in the string being searched:

```
>>> str5 = 'string to be 445\t3665 searched for phone number'
>>> print(str5)
string to be 445        3665 searched for phone number
>>> match = re.search( '\d\d\d\s\d\d\d\d', str5)
>>> if match: print (f'found the pattern: {match.group()}')
```

Q: Did the tab match against the same pattern?

Create a working regular expression pattern to match a phone number in the following format, from the following string:

```
'string to be (0)131 445-3665 searched, for phone number'
```

Q: What was the working pattern?

4. Repeating Pattern Characters

Often in a pattern we want to look for a sequence of several characters of the same type.

- **+** matches one or more characters
- ***** matches zero or more characters
- **?** matches zero or one characters

Try:

```
>>> str7 = 'reeeeeeeeeeepeating'
>>> match = re.search( 'e+', str7)
>>>
>>> if match: print (f'found the pattern: {match.group()}')
```

Q: What has been matched against the pattern?

Sometimes we know how many characters are in parts of a pattern.

Try:

```
>>> str8 = 'Bill B Phone no:07284739846 age:57'
>>> match = re.search( '\d{11}', str8)
>>>
>>> if match: print (f'found the pattern: {match.group()}')
```

Q: What has been matched against the pattern?

Sometimes we do not know how many characters are in part of a pattern.

Try:

```
>>> str9 = 'there are 7284739846 students in the uni'
>>> match = re.search( '\d+', str9)
>>>
>>> if match: print (f'found the pattern: {match.group()}')
```

Q: What has been matched against the pattern?

Often whitespace or separating characters are sometime there and sometimes not. That is where the special * pattern matching character can be useful.

Try:

```
>>> str10 = '7 8'
>>> match = re.search( '\d\s*\d', str10)
>>>
>>> if match: print (f'found the pattern: {match.group()}')
```

Q: What has been matched against the pattern?

With the same pattern try matching against the strings:

```
>>> str11 = '7  8'
>>> str12 = '7     8'
>>> str13 = '78'
```

Q: Did the pattern successfully match all the strings?

Q: Why does it match the last string?

Often when performing digital investigations, searches for phone numbers, email addresses, and custom patterns for other forensic artefacts need to be carried out in the content of evidence files. Regular expressions are excellent for this type of work.

Create a working regular expression pattern to again match on the phone number from earlier, but this time use repeating characters to match the pattern of the phone no in this string:

```
'string to be (0)131 445-3665 searched for phone number'
```

Q: What was the working pattern?

5. More Special Characters

There are many special pattern matching characters. Below is a small selection of common, and useful ones.

- **\t** matches tab character
- **\n** matches newline character
- **\S** matches non whitespace characters
- \ escapes a special pattern character.
- **\\.** matches on an actual dot, not any character.
- **Anchors:**
  - **^** matches **position** at the beginning of a string (does not match a char only the position)
  - **$** matches **position** at the end of a string (does not match a char only the position)

Sometimes we want to escape special characters if we are looking for those literal characters. For example, to match an actual '.' (dot) character, we need to escape it using a **\\.** Try:

```
>>> str14 = 'The DoS came from: 146.176.164.343 in the C27 lab'
>>> match = re.search('\d+\.\d+\.\d+\.\d+', str14)
>>>
>>> if match: print (f'found the pattern: {match.group()}')
```

Q: Did the pattern successfully match the IP Address?

Create a working regular expression pattern to match on email addresses using repeating characters to create the pattern. Test it on:

```
'string john@cleese.com to be searched for email address'
```

Q: What was the working pattern?

Q: Can you explain every character in your regex working pattern? (this level of understanding will be needed for the assessment)

When parsing log files or output from command line tools, often the first field in a line is where the pattern we are looking for is.

Try:

```
>>> str15 = 'host: 146.166.55.2: net scanning against 146.156.12.2'
>>> match = re.search('host:\s\d+\.\d+\.\d+\.\d+', str15)
>>>
>>> if match: print (f'found the pattern: {match.group()}')
```

Q: Did the pattern successfully match the host?

However if there was no "host: " pattern to use to find the record we are looking for, but the IP address was at the start of every line (which is common), we could use the **^** special character to anchor our search to the start of the string.

```
>>> str16 = '146.166.55.2: host net scanning against 146.156.12.2'
>>> match = re.search('^\d+\.\d+\.\d+\.\d+', str16)
>>>
>>> if match: print (f'found the pattern: {match.group()}')
```

Q: Did the pattern successfully match the host at the start of the string?

Q: Write a regex to match the IP address at the END of the string only. What is the regex used?

6. Sets and ranges of characters with [ ] (square brackets)

Sets of characters

Different single characters are commonly used in patterns, such as phone number separator characters. Data often has interchangeable characters and we may want to match on any of these.

Try:

```
>>> str17 = 'string to be (0)131 445 3665 searched, for phone number'
>>> match = re.search('\d+[- ]\d+', str17)
>>>
>>> if match: print (f'found the pattern: {match.group()}')
```

Q: What has been matched against the pattern?

Extend the pattern to match an entire phone number, so it matches all of the following:

```
>>> str21 = '(0)131-445-3665'
>>> str22 = '(0)131 445-3665'
>>> str23 = '01314451633'
```

Q: Did the pattern successfully match all the strings?

Q: What was the working pattern?

Improve your email address regular expression pattern from earlier in the lab, to match on all the following email addresses, using optional characters in the pattern:

```
'john@cleese.com'
'john.cleese@montypython.com'
'michael-palin@python.co.uk'
```

Q: What was the working pattern?

7. Ranges of characters using [] (square brackets)

Ranges of characters are also commonly used in patterns. The – (hyphen) can be used within [] to specify ranges of numeric or alphabetic characters.

Try:

```
>>> str24 = "Rich's password:aPPl3s345 is strong"
>>> match = re.search('[a-z]:[a-zA-Z0-9]+', str24)
>>>
>>> if match: print (f'found the pattern: {match.group()}')
```

Q: What has been matched against the pattern?

Create a pattern to match any of the following napier web servers:

```
>>> str31 = '146.176.123.2'
>>> str32 = '146.176.123.8'
>>> str33 = '146.176.123.9'
```

Q: Did the pattern successfully match all the strings?

Q: What was the working pattern?

If your pattern was 146.176.123.[0-9], this matches only a single digit after the last dot. So, to match on all IP Addresses on the server subnet (0-255) we need to use a different technique.

[0-255] could not be used as the pattern as a square bracket only ever represents ONE character - [0-255] would be interpreted as 0,1,2 or 5.

To create a pattern to match a range of the napier web servers, we can use multiple ranges:

To match on `'146.176.123.2' or '146.176.123.99'`
Try:

```
>>> str34 = 'server 146.176.123.78'
>>> match = re.search('146\.176\.123\.[0-9][0-9]', str34)
>>>
>>> if match: print (f'found the pattern: {match.group()}')
```

Q: What has been matched against the pattern?

Match the pattern against the following napier web servers:

```
>>> str35 = '146.176.123.15'
>>> str36 = '146.176.123.2'
```

Q: Did the pattern successfully match all the strings?

Q: What was the problem?

To create a pattern to match the full range of the napier web servers with a single or two digits, we can use the "?" (zero or one occurrences)

To match on `'146.176.123.0' - '146.176.123.99'`
Try:

```
>>> strx = 'server 146.176.123.8'
>>> match = re.search('146\.176\.123\.[0-9][0-9]?', strx)
>>> if match: print (f'found the pattern: {match.group()}')

>>> strx = 'server 146.176.123.88'
>>> match = re.search('146\.176\.123\.[0-9][0-9]?', strx)
>>> if match: print (f'found the pattern: {match.group()}')
```

Q: Can the pattern match all the servers in the range 0-99?

Q: How would you change the pattern to match the full range 0-255?

8. Options – using the or | symbol

To specify several alternative patterns (or sub-patterns) to be matched, we can use the OR (or alternation) operator |. The vertical line is at the bottom left of my keyboard.

To see the or operator working try the following:

```
>>> str44 = 'href="https://www.blah.com"'
>>> match = re.search('http|https|ftp', str44)
```

```
>>>
>>> if match: print (f'found the pattern: {match.group()}')
```

Q: What has been matched against the pattern?

Q: How would we fix this problem?

Q: Explain your regular expression in detail.

9. Groups – with ( ) round brackets

Groups can be used to divide a regex into sub-expressions, and to extract specific parts of the regex pattern. A Regular Expression pattern can be used to match against a large pattern, and smaller parts of the matched text can be extracted with groups. In our first example, we use groups to show how you could extract the name and the domain parts of an email address.

Try:

```
>>> e1 = 'johncleese@montypython.com'
>>> match = re.search('(\w+)@(\w+.com)', e1)
>>>
>>> if match: print (f'found the pattern: {match.group()}')
```

Q: What has been matched against the entire pattern?

Now check what has been returned as a match to group 1:

```
>>> match.group(1)
```

Q: What has been matched against group 1 part of the pattern?

Now check what has been returned as a match to group 2:

```
>>> match.group(2)
```

Q: What has been matched against group 2 part of the pattern?

This can be very useful if you have to match on a large pattern, but want to get hold of parts of it for further processing.

10. Finding All Matches in a String

The **re.search()** method we have used so far finds only the first match of a pattern. This could be used repeatedly, such as on each line of a file we are analysing, but often it may be more useful to find all matches at once. The **re.findall()** method performs this task.

Create a string with several emails addresses in it:

```
>>> e2 = 'johncleese@montypython.com blah rich@napier.com blah'
```

Search the string with an email pattern:

```
>>> match = re.search('\w+@\w+.com', e2)
>>>
>>> if match: print (f'found the pattern: {match.group()}')
```

Q: What has been matched against the pattern?

Q: What type of object has been returned?

Now use **findall()** to search the string with the email patterm:

```
>>> matches = re.findall('\w+@\w+.com', e2)
>>>
>>> print (matches)
```

Q: What has been matched against the pattern?

Q: What type of object has been returned?

re.findall() returns a list which can be iterated over, or used as input to another task.

11. findall() and Groups

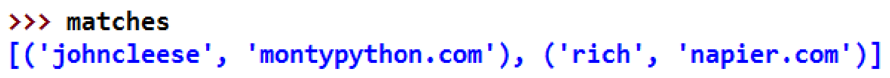
Groups can be combined with findall() to return groups for each match for all matches.

```
>>> matches = re.findall('(\w+)@(\w+.com)', e2)
>>>
>>> print (matches)
```

Q: What has been matched against the pattern?

Q: What type of objects have been returned in the list?

A list of tuples is returned, with the two groups being returned as the two parts of a tuple.



```
>>> matches
[('johncleese', 'montypython.com'), ('rich', 'napier.com')]
```

If we were looking for only the domains (which might be handy if we were doing some reconnaissance as part of some security testing), we can get hold of them from the 2nd part of each tuple:

```
>>> for match in matches: print (match[1])
montypython.com
napier.com
```

12. Reading content from web pages

This exercise will feed directly into to the final coursework assessment.

To retrieve data from the web, we can use the Python standard module urllib.request to access and read data via a URL argument. From the IDLE Shell, using **File>New Window**, create a script **webpage_get.py**, based on the code below:

You can copy&paste the code from the link below to create the initial script.

Script starting point: webpage_get_start.py (in moodle)

```
# Script:   webpage_get.py
# Desc:     Fetches data from a webpage.
# Author:   PL & RM
# Modified:  Oct 2017
#
import sys, urllib.request

def wget(url):
    ''' Retrieve a webpage via its url, and return its contents'''
    print ('[*] wget()')
    # open url like a file, based on url instead of filename
    webpage = None # ADD YOUR CODE TO OPEN URL HERE
    # get webpage contents
    page_contents = None # ADD YOUR CODE HERE
    return page_contents

def main():
    # specify test url
    sys.argv.append('http://www.napier.ac.uk/Pages/home.aspx')

    # Check args
    if len(sys.argv) != 2:
        print ('[-] Usage: webpage_get URL')
        return

    # Get web page
    print (wget(sys.argv[1]))

if __name__ == '__main__':
      main()
```

Add code to use urllib.request to create a webpage object and then read the contents from it. Test from the IDLE shell, by running the module using **Run>Run Module (or F5).**

Hint: see the lecture slides for reference.

Q: Does the code successfully fetch the contents of the Napier web page?

Q: what specific type of string are the page contents returned as? (check the beginning of the returned string)

13. Email header analysis

Let's pull together some of the concepts covered in today's lab (and of course earlier ones too!)

Create a module called **email_analysis.py**. You can copy&paste the code from the link below to create the initial script. This should run, but won't do much yet!

Script starting point: **email_analysis_start.py** **(in Moodle)**

The script should run without errors, but not do too much yet!

```
# Script:   email_analysis.py
# Desc:     extracts email addresses and IP numbers from a text file
#           or web page; for example, from a saved email header
# Author:   Petra Leimich Oct 2017
#
# IMPORTANT: wget may fetch a byte object, but regex only works with strings
import sys, urllib.request, re
# import webpage_get #imports your own webpage_get module. (works best if file
is in Python36 folder)

# if you don't want to import your existing module, you need to define wget
here:
def wget(url):
    '''Suitable function doc string here'''
    # open url like a file, using url instead of filename
    # then get webpage contents and close
    # ... ADD YOUR CODE HERE ...
    return page_contents

def txtget(filename):
    '''Suitable function doc string here'''
    # open file read-only, get file contents and close
    # ... ADD YOUR CODE HERE ...
    return file_contents

def findIPv4(text):
    '''Suitable function doc string here'''
    ips = [] # ... ADD YOUR CODE HERE ...
    return ips

def findemail(text):
    '''Suitable function doc string here'''
    emails = [] # ... ADD YOUR CODE HERE ...
    return emails

def main():
    # url argument for testing
    # un-comment one of the following 4 tests at a time
    #sys.argv.append('http://www.napier.ac.uk/Pages/home.aspx')
    sys.argv.append('http://asecuritysite.com/email01.txt')
    #sys.argv.append('http://asecuritysite.com/email02.txt')
    #sys.argv.append('email_sample.txt')
    # Check args
    if len(sys.argv) != 2:
        print ('[-] Usage: email_analysis URL/filename')
        return

    # Get and analyse web page
    try:
        # call wget() or txtget() as appropriate
        # ... ADD YOUR CODE HERE ...
        print ('[+] Analysing %s' % sys.argv[1])
        print ('[+] IP addresses found: ')
        # ... ADD YOUR CODE HERE ...
        print ('[+] email addresses found: ')
        # ... ADD YOUR CODE HERE ...
    except:
        # error trapping goes here
        pass # ... ADD YOUR CODE HERE ...

if __name__ == '__main__':
      main()
```

Your task is to complete the script so that it fulfils the requirements listed below. Use the lecture notes to help you get started.

Remember that when you read a webpage, the contents are typically extracted as a byte array (binary string) rather than a string. You will need to decode this so that regex can get to work.

The script should:

- Read the contents of a text file or URL, as given by sys.argv[1] (typically, this could be a stored email header).
- Use regular expressions to extract all IPv4 addresses and email addresses from the contents, and then print the results
- Through exception handling, deal with file/url not found errors and situations where no IP addresses and/or no email addresses are found – ideally, this should provide specific messages.
- Use of the start script is very strongly recommended.
- Enhancements:
    - Auto-detect whether the argument is a URL or local file, and automatically call the appropriate function to open and read the object
    - IP addresses must have a number between 0-255 in each quartet (0.0.0.0-255.255.255.255)

- Email headers typically contain multiple occurrences of email addresses. Ideally, your script should count the number of occurrences and print each email address only once with its count, instead of repeating email addresses found more than once.

Test your script with email_sample.txt (moodle), http://asecuritysite.com/email01.txt, http://asecuritysite.com/email02.txt.

14. Discussion of the email analysis script

Thinking about these two questions will increase your understanding of this case study script.

a) You will have noticed that some Message IDs are formatted like email addresses, for example, 20130601020815.EC2BDC477EA07111@callnet.dk in http://asecuritysite.com/email01.txt. Discuss whether it would be possible to change the script to remove these from the list of email addresses found. If yes, propose a suitable algorithm / pseudocode. [You are NOT asked to implement this!]

b) Email_sample.txt contains some IPv6 addresses. Why would it be difficult (or almost impossible) to extract a list of all IPv6 addresses using regex? Hint: Read the short section "Recommended representation as text" of https://en.wikipedia.org/wiki/IPv6_address if you are not familiar with the possible formats of IPv6.

References:

Interactive RegEx tester for Python: http://pythex.org/.

Interactive Regex tester (make sure you select the Python option): http://regex101.com

Interactive RegEx tutorial for Python: https://regexone.com/

15. OPTIONAL Challenge exercise

Part 1

Most UK postcodes have the pattern "one or two upper case letters followed by one or two digits, a space, one digit and two upper case letters". Write a regular expression to match this, and test it on the following strings:

```
pk1 = 'Edinburgh Napier Merchiston postcode EH10 5DT'
pk2 = 'Abbey Road- THE No1 1st zebra crossing - is at NW8 9AY'
pk3 = 'Glasgow Queen Street Station G1 2AF'
```

If your regex found the postcodes in all of the above, and correctly did not identify No1 1st as a postcode, you are ready to move on.

Part 2

Unfortunately, some London postcodes have a digit and a letter at the end of the first group rather than two digits, for example:

```
pk4 = 'Buckingham Palace SW1A 1AA is an example of a letter at the end of
the first quartet'
```

Test your current pattern with pk4. Does it match?

Now change your pattern to account for this new information and re-test it on pk4. The first part is still only allowed to be length 4: Make sure that your answer does not match pk5:

```
pk5 = 'SW25X 1AA is not a valid pattern and should not be matched'
```

Note that we have still simplified things a bit, as not all possible letter and number combinations with our pattern are in use. For a postcode validator or online address look up feature, we would want to exclude postcodes that don't currently exist, for example, 'ZZ23 1AA'. If you really want to see the full and gory detail, try https://en.wikipedia.org/wiki/Postcodes_in_the_United_Kingdom. A proposed RegEx pattern can be found at http://regexlib.com/REDetails.aspx?regexp_id=260.

Remember, use patterns found on the internet only if you fully understand them and if you have personally tested them comprehensively!